# *SZAKDOLGOZAT*

Tar Zoltán

Debrecen

2009

# Debreceni Egyetem

# Informatika Kar


## Webalkalmazások fejlesztése Ruby on Railsel


**Téma vezető:**

**Espák Miklós**

**Egyetemi tanársegéd**

**Készítette:**

**Tar Zoltán**

**Programtervező Informatikus**



**Debrecen**

**2009**

# Tartalomjegyzék

1. Bevezetés....2
2. A Rails megközelítése és filozófiai alapja....4
2.1 A Rails agilis szemlélete....4
2.2 MVC....5
2.3 MVC és a Rails....6
2.3.1 Active Record, a Rails modell megfelelője....6
2.3.2 Action Pack: Nézet és Vezérlő....7
2.4 REST....8
3. Ruby mint programozási nyelv....10
3.1 Ruby eredete....10
3.2 Típusok, alaposztályok....10
3.3 Modulok és osztályok....13
3.4 Vezérlés és elnevezési konvenciók....16
3.5 Ruby segédprogramok....17
4. Ruby on Rails részletesen....19
4.1 A Ruby on Rails eredete....19
4.2 Naplózás és dokumentálás....19
4.3 Rails könyvtár struktúra....20
4.4 Migrációk....22
4.4.1 Migrációk felépítése....23
4.4.2 Adat migrációk....25
4.5 Active Record....26
4.6 Táblák közötti kapcsolat....29
4.7 Validáció....31
4.8 Action Controller....33
4.8.1 Szűrők....34
4.9 Action View....34
4.9.1 Helper....35
5. Esettanulmány, Black Market....38
5.1 Alkalmazás feladata és célja....38
5.2 Adatbázis kapcsolatok és felépítés....38
5.3 Felhasználói jogkörök és beléptetés....39
5.4 Aukció menete....40
5.5 Alkalmazás lokalizációs beállítása....41
6. Összefoglalás....42

# 1. Bevezetés

Az World Wide Web 1989-ben jött létre a Genovaban a CERN falai között, az internet mára szinte mindenki számára elérhető. Kezdetben az internet a kutatók közötti információ, dokumentumok megosztására szolgált. A korai oldalak egyszerűek, jól átláthatóak voltak, az idő múltával sok új lehetőség, technológia született ami az internetes oldalakat se hagyta nyom nélkül. Az új technológiák beépülésével az oldalak komplexitása megnövekedett és külön a webes alkalmazásokra létrehozott keretrendszerekkel készülnek a mai webalkalmazások.

Az internet egyre szélesebb körű elterjedésével, sokkal több információ, alkalmazás érhető el mint akár 10 éve. Kezdetben csak statikus tartalmak, leírások, dokumentációk voltak elérhetők. Ma a leglátogatottabbak az ergonomikus, dinamikus tartalommal rendelkező oldalak, mint például a videómegosztók, közösségi oldalak, webáruházak.

A Ruby on Rails egy olyan keretrendszer amivel webalkalmazások fejlesztése és karbantartása lényegesen egyszerűbb, gyorsabb. Sok cég, vállalkozás készítettet webalkalmazásokat, legfőképp Java, PHP vagy .NET technológia segítségével. A fejlesztések során kiderült hogy az eddigi módszerek nehézkesek és néhány pontra nem helyeznek elég hangsúlyt. Az új igények hatására új szemléletek, metodikák jöttek létre. A Rails ezeket próbálja ötvözni, mint a web alapú keretrendszerek egyik legújabb tagja.

A Rails fejlesztésében sok ember vesz részt és népes felhasználó bázisa van sok on-line fórummal, az informatika nagy alakjai is említést tesznek róla:

"It is impossible not to notice Ruby on Rails. It has had a huge effect both in and outside the Ruby community... Rails has become a standard to which even wellestablished tools are comparing themselves to." *Martin Fowler*

Szakdolgozatommal azt szeretném bemutatni hogy a Rails keretrendszer segítségével mennyivel leegyszerűsödik a fejlesztés, a webalkalmazások gyorsabban jönnek létre, vázuk és az alapvető funkciók leegyszerűsítésével több idő és energia jut a tartalom, funkciók bővítésére és hogy ez hogy valósítható meg.

Szakdolgozatomban a Rails összetevőit, felépítést, a filozófiai hátterét fogom

kifejteni, illetve a Ruby programozási nyelv alapjait ismertetném hiszen enélkül nem lehetséges a Rails alkalmazást készíteni, néhány résznél példa kódokkal is szerepelnek.

Szakdolgozatom utolsó részében a saját alkalmazáson szemléltetem hogy mely eszközöket használtam, illetve hogy mi a volt az elvi felépítése. Egy gyakorlati alkalmazáson jobban látszódik hogy  a mai kor követelményeire milyen eszközöket ad a Rails mint keretrendszer.

# 2. A Rails megközelítése és filozófiai alapja

A Rails egy komplex keretrendszer, amit Model-View-Controller architektúra alapján épül fel, amiben minden kódrészletnek megvan a maga helye és célja. Rails esetében nagy hangsúly van a tesztelésen, ehhez a szükséges eszközöket biztosítja.

A Rails alkalmazásokat Ruby nyelven íródnak ami egy modern, objektum orientált szkript nyelv. A Ruby segítségével rövid, tömör, jól olvasható kódot írhatunk, amelyet akár hónapok múlva is megértünk.

Filozófiailag is meg van alapozva a Rails szemlélete, melynek két fő eleme a DRY és COC. A DRY (don't repeat yourself) azt hangoztatja hogy egy alkalmazásban ne ismételjünk az információkat fölöslegesen, minden lehetőleg legyen a neki fenntartott helyen. Így ha esetleg módosítanánk az alkalmazást akkor azt csak egyetlen egy helyen kelljen módosítani.

A COC (Convention over configuration) a konvenciókat helyezi a konfiguráció elé, alapértelmezetten paraméterek többsége be van állítva, ezért sokkal kevesebb időt kell konfigurálásra szánni mint például egy Java alkalmazásnál, de ha kell akkor a Rails biztosít eszközöket amelyekkel könnyen paraméterezhetők az alkalmazások.

A fejlesztőknek egyszerű AJAX vagy REST kódokat az alkalmazásaikba építeni, mert alapértelmezetten támogatja a Rails.

## 2.1 A Rails agilis szemlélete

A Rails megalkotásánál fontos volt hogy az agilis eszközöket és metódusokat is felhasználjanak, amit eddig csak nagyvállalatok használtak legfőképp, de a kis és közepes cégeknél is terjed.

Az agilis módszer tulajdonságai:

- Az egyének és interakciók előtérbe helyezése a folyamatok és az eszközök ellenében.
- Működő szoftver előtérbe helyezése a mindent átfogó dokumentáció ellenében.
- Az ügyféllel való együtt működése előtérbe helyezése a szerződésben foglaltak ellenében.

- A változásokra való reagálás előtérbe helyezése tervkövetés ellenében.

A Rails hangsúlyozza az egyének és interakciók fontosságát, előnyben részesíti a kis csoportban dolgozó fejlesztőket, ami átláthatósághoz vezet hiszen amit a fejlesztők csinálnak az azonnal látszik a megrendelő számára, az így létrejövő interakció fontos része a fejlesztésnek.

A Rails nem tagadja a dokumentáció hasznosságát, viszont nem arra épül, habár a Rails lehetőséget ad hogy az egész kódot lefedő HTML dokumentációt hozzunk létre. A kód egyértelműsége illetve hogy egész hamar működő kód jön létre határozza meg a fejlesztést.

A Rails támogatja a megrendelővel való együttműködést, hiszen egy Rails projekt tud gyorsan alkalmazkodni az igényekhez és amikor látja a megrendelő a fejlődést, a bizalom megmarad a projektben.

A változás minden projektben fellelhető amit kisebb-nagyobb sikerrel tudnak alkalmazkodni, erre a jelenségre ad módszert a Rails hiszen DRY elv alapján, minden módosítást csak egy helyen kell végrehajtani.

Ahelyett hogy a Rails görcsösen próbálná az agilis elveket átvenni, megpróbál annak lényegéből meríteni hogy a megrendelővel, felhasználóval együttműködve készüljön a projekt. A párbeszéd legyen döntő, a változásokra fellépő helyes lépés meghozatala a megrendelővel, felhasználóval együtt történjen meg.

## 2.2 MVC

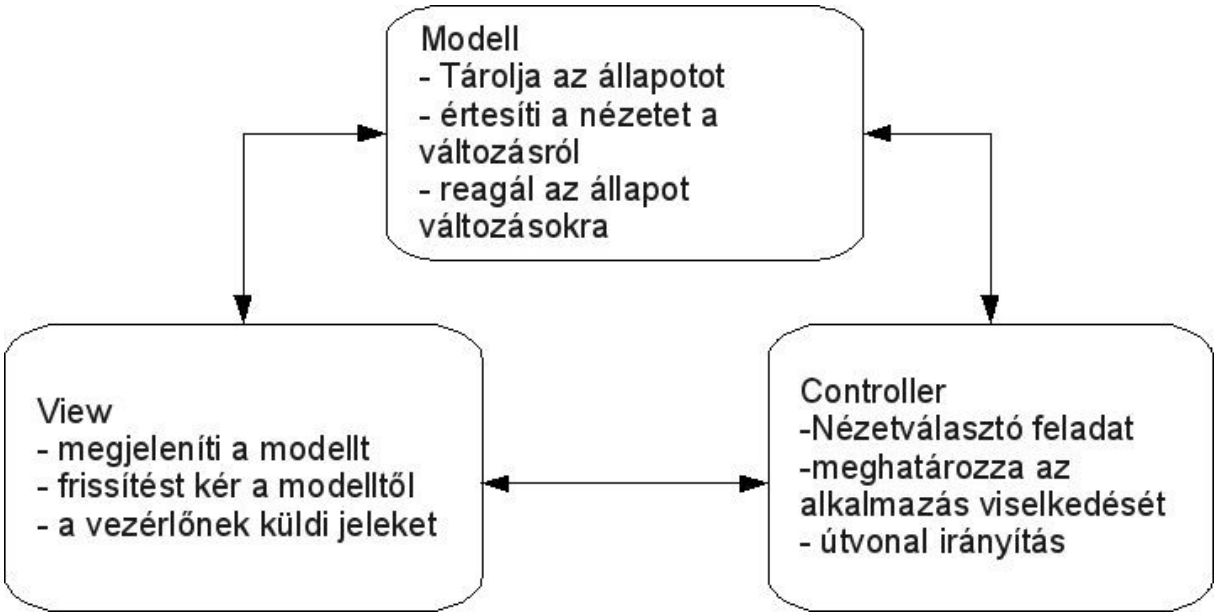
1979-ben Trygve Reenskaug, egy új architektúrát fejlesztett ki interaktív alkalmazásokhoz, ahol az alkalmazást 3 részre bontja: modell, nézet, vezérlő.

A Design Patterns: Elements of Reusable Object-Oriented Software című könyv is leírja az MVC felépítését. A modell felelős az állapot kezeléséért az alkalmazásban. Az állapot lehet tranziens amikor egy állapot csak néhány művelet erejéig érvényes vagy lehet perzisztens amikor az állapotot egy külső helyen tároljuk általában egy adatbázisban. A modell felelős az üzleti logikáért is, itt van implementálva, ez azért jó ha az adott adat és az üzleti logika egy helyen van akkor biztos hogy máshol nem rontjuk el.

A nézet adja a felhasználó felé a felületet, amit a felhasználó látni fog, amit általában a modell alapján tesz meg. A nézet a kapocs a a modell és a felhasználó között.

A vezérlő határozza meg hogy külső beavatkozás során mi történjen az alkalmazásban. Ez a külső beavatkozás általában adat bevitel, melynek hatására módosul a modell és az megfelelő nézetet továbbítja a felhasználó felé.

Az MVC modellt eredetileg szokványos grafikus felülettel rendelkező alkalmazásokhoz szánták, mára leginkább webes alkalmazások körében használják. A MVC modell lehetővé tette hogy a kód egyszerűbben előállítható és karbantartható legyen, mint egy egyszerű monolitikus felépítésű kód.




*ábra 1: Az MVC modell felépítése*

## 2.3 MVC és a Rails

### 2.3.1 Active Record, a Rails modell megfelelője

Az Active Record a Rails Objektumrelációs leképzés(ORM) implementációs rétege. A szabványos ORM leképzést szorosan közelíti, a táblákat az osztályokhoz, a sorokat az objektumokhoz és az oszlopokat az objektum attribútumaihoz. Amiben eltér a legtöbb ORM implementációtól az a konfiguráció, az Active Record próbálja minimalizálni a fejlesztők konfigurálási munkáját. Az Active Record fejlett validálási módszerekkel rendelkezik hogy a modell adatait ellenőrizhessük.

### 2.3.2 Action Pack: Nézet és Vezérlő

A vezérlő és a nézet közeli kapcsolatban van, a vezérlő látja el adatokkal a nézetet és a vezérlő a nézettől kapja az interakciókat. A közeli kapcsolat miatt a Rails egy egység alá vonta őket, Action Pack néven.

#### Nézet megvalósítás

A nézet határozza meg hogy a böngészőben hogy jelenik meg egy oldal. A legegyszerűbb HTML kóddal is meg lehet valósítani, de általában dinamikus tartalmat szeretnénk megjeleníteni. Rails estében a dinamikus tartalmat három fajta sablon alapján lehet létrehozni. A leggyakrabban használt sablon az ERb (Embedded Ruby), aminek segítségével Ruby kód részleteket tehetünk a nézetbe, amely flexibilis, de néhány esetben áthágja az MVC szemléletet, hiszen ha a nézetben lévő kód inkább a modellben vagy a vezérlőben lenne a helye.

XML Builder-t lehet használni hogy XML dokumentumot hozzunk létre Ruby kód segítségével. RSJ nézet, ennek segítségével Javascript kód részleteket hozhatunk létre a szerveren amelyek majd a böngészőn fognak lefutni. Ennek segítségével hozhatunk létre AJAX felületet.

#### Vezérlő

A Rails vezérlő a tényleges irányítója az egész alkalmazásnak. A bekövetkezett változás alapján kommunikál a felhasználóval, a nézettel és modellel. Az irányító a háttérben dolgozik, így lehetőséget ad hogy alkalmazás szintű funkcionalitásra koncentráljunk a kód írásánál. A vezérlő több lényeges funkciót lát el:

Szabályozza az URI címek értelmezését, hogy a megfelelő URI-hez a megfelelő funkcionalitás társuljon. így felhasználó barát címzés hozható létre, gyorsítótár alkalmazását is itt lehet befolyásolni ezzel teljesítménynövekedést érhetünk az alkalmazásban.

Segítő modulokat módosíthatunk, hozhatunk létre, amivel elkerülhetjük hogy a nézet sablonunkba kelljen azt beletennünk, ezzel jobban elkülöníthetjük a kódunkat, a munkameneteket is a vezérlő segítségével szabályozhatjuk.

## 2.4 REST

A Restful Web Services című könyv amit Leonard Richardson, Sam Ruby írt leírja hogy mi a REST alapja és történetét. Egy erőforráshoz egy cím tartozik, ahonnan az erőforrást letöltve a HTTP metódustól függően azt lekérhetjük, létrehozhatjuk, frissíthetjük, vagy törölhetjük. A HTTP protokoll esetében ezt a szerepet az URI tölti be. A REST elve ugyanakkor a HTTP metódusok – GET, POST, PUT, DELETE - szigorúbb használatával leegyszerűsíti az eléréseket. Ezt először Roy Fielding hangsúlyozta ki a webalkalmazások erőforrás szemléletét, és ennek az architektúrális paradigmájának a REST nevet adta.

| | Hagyományos | | REST | |
|---|---|---|---|---|
| Művelet | Metódus | Cím | Metódus | Cím |
| Listázás | GET | http://test.eu/products | GET | http://test.eu/products/index |
| Lekérés | GET | http://test.eu/products/1 | GET | http://test.eu/products/show?id=1 |
| Létrehozás | POST | http://test.eu/products/1 | GET | http://test.eu/products/create |
| Frissítés | GET | http://test.eu/products/1 | GET | http://test.eu/products/update?id=1 |
| Törlés | DELETE | http://test.eu/products/1 | GET | http://test.eu/products/delete?id=1 |

A Rails próbálja a REST elvet minél inkább beépíteni és használni, a fentiek kiegészítésre még használ 2 metódussal bővítette ez az ***edit*** és a ***new***.

| Művelet | Metódus | Cím | Vezérlő | Esemény | ID paraméter |
|---|---|---|---|---|---|
| Listázás | GET | http://test.eu/products | products | index | Nincs |
| Lekérés | GET | http://test.eu/products/1 | products | show | 1 |
| Létrehozás | POST | http://test.eu/products/1 | products | create | nincs |
| Frissítés | PUT | http://test.eu/products/1 | products | update | 1 |
| Törlés | DELETE | http://test.eu/products/1 | products | delete | 1 |
| Új elem adatainak megadása | GET | http://test.eu/products/new | products | new | nincs |
| Meglévő elem új adatainak megadása | GET | http://test.eu/products/1/edit | products | edit | 1 |

A REST előnyei közé tartozik hogy az URI-ek tisztábbak lesznek és Rails-es REST vezérlőkkel egyszerűen lehet ugyanarra az erőforrásra a kéréstől függően különböző formátumokkal válaszolni.

# 3. Ruby mint programozási nyelv

## 3.1 Ruby eredete

A Ruby nyelvet 1995-ben Yukihiro Matsumoto készítette el, sokáig kereste az ő igényeinek a legmegfelelőbb szkript nyelvet, de nem találta meg, a Perl funkcionalitása nem volt elég és a Python pedig nem volt elég objektumorientált. A Ruby Japánban kezdett el először elterjedni, majd a Rails keretrendszer elterjedésével a Ruby felé is megnőtt az érdeklődés. Főbb tulajdonságai:

- Általános célú szkript nyelv
- Open Source, minden operációs rendszeren futtatható
- Dinamikusan típusos, teljesen objektum orientált, minden osztálypéldány
- Nincs változó deklaráció, csak definíció
- Perl, Python, Smalltalk azok a nyelvek amelyekből sokat merített

## 3.2 Típusok, alaposztályok

A Ruby-ban nincsenek igazi típusok csak alaposztályok. Ezek az alaposztályok a Number (szám), String (szöveg), Symbol (szimbólum), Array (tömb), Hash (táblázat), Range (sorozat/tartomány) és a RegExp (reguláris kifejezés).

| Típus | Osztály | Jelölés |
|---|---|---|
| Szám | Fixnum vagy Bignum | 24 vagy 2_000_000 |
| Szöveg | String | 'Hello World!' |
| Szimbólum | Symbol | :szimbólumNeve |
| Tömb | Array | [ 'Alma', 42, :szimbolum ] |
| Táblázat | Hash | {:kulcs => ertek } |
| Sorozat, Tartomány | Range | ('a' .. 'z' ) vagy (1..100) |
| Reguláris kifejezés | RegExp | /reguláris kifejezés/ |

## Adattípusok

A Rubyban a típusok futási időben rendelődnek hozzá a változókhoz, de ezt nem jelenti hogy ne lennének a primitív típusosok támogatva. A Rubyban az alábbi primitív típusok vannak:

## Number

A Number típus az egész számok ábrázolására szolgál. Ezek lehetnek pozitív vagy negatív előjelűek. A számon belül lehetséges elhatároló jelet alkalmazni a könnyebb olvashatóság miatt. Például:

```
szam: 5000
```

vagy

```
nagy_szam_elhatarolva: 5_000_000
```

## Float

Minden nem egész szám float típusú. A float típusú számokat meg lehet adni tudományos jelek segítségével vagy hagyományos tizedesponttal is. A float számokat így lehet megadni:

```
pi: 3.14
```

vagy

```
atmero: 3.14e-05
```

## String

Minden alfanumerikus karakterlánc amelyet idézőjelek határolnak String típusúak. Mindkét idézőjel típus érvényes használatnál. Például:

```
szoveg: ''Hello, ez egy String!''
```

vagy

```
szoveg2: 'Hello, ez is egy String!'
```

## Tömbök

A Rubyban a tömbök indexelt kollekciók. Automatikusan növekedik ahogy újabb elemeket tárolnak. A tömbök elemihez gyorsabban férhetünk hozzá míg a táblázatok több rugalmasságot biztosítanak. Lehetőség van arra hogy az olyan tömböket amik csak stringeket tartalmaznak hogy egy speciális formába adjuk meg a **%w** operátor segítéségével. Bármelyik tömb vagy táblázat tartalmazhat különböző típusú elemeket, például lehetséges hogy egy tömbnek number, string vagy float legyen az egyik eleme.

```
valtozo =  42
tomb = [valtozo, 2, "Word"]
tomb2 = %w(alfa beta omega) #tomb2=["alfa","beta","omega"]
```

## Hash

Hash táblát úgy definiálhatunk vagy hozhatunk létre ha az elemeket megadjuk [] között. A hash táblák hasonlóak a tömbökhöz csak itt {} jelek állnak és minden bejegyzés 2 részből áll, egy kulcs részből és a hozzárendelt értékből, a kettőt => választja el. A kulcsnak egyedinek kell lennie. A kulcs és az érték tetszőleges, lehet tömb, egy másik táblázat, stb. Rails esetében a kulcsok tipikusan szimbólumok. Ha esetleg egy olyan kulcsra hivatkozunk amely nem létezik akkor a NIL-t kapunk vissza.

```
hash = {"kulcs1" => "érték", :kulcs2 => "érték2", :kulcs3
     => "érték3"}
```

## Range

A range objektumok sorozatok amit két módon is megadhatunk , az egyikben két a másikban három darab pont van a szélső elemek között. A két pontos változatba a záró elem is beletartozik, a három pontos változatban az kimarad. Mivel a nyelvből hiányzik a hagyományos előírt lépésszámú ciklus, a sorozatok leginkább ilyenek készítésére

használatosak.

```
(4..78).each do |i|
     puts i
    end
```

## Reguláris kifejezések

A reguláris kifejezéseket mintaillesztésre használják, ennek az osztálya a RegExp. Ruby mintaillesztési mechanizmusa a Perl nyelvéhez hasonlítható. A mintaillesztésnek külön operátora van ez a =~ , aminek a bal oldalán a reguláris kifejezés áll a jobb oldalán pedig az adott karakterlánc. A visszatérési érték lehet NIL ha nem illeszkedik illetve ha illeszkedik akkor megadja hogy hanyadik karakternél illeszkedik a minta.

## Konverziók

A Ruby változók értékadáskor veszik fel osztályukat, de ezután erősen típusos változóként viselkednek, tehát nem lehet egy stringet számként vagy egy számot stringként használni. Ehhez konvertáló metódusok állnak rendelkezésre. Stringgé alakításhoz a `.to_s` , a számmá alakításhoz a `.to_i` metódust használjuk.

# 3.3 Modulok és osztályok

## Modul

A modulok abban hasonlítanak az osztályokra hogy metódusokat, konstansokat és más modult, osztályt is tartalmazhatnak, de az osztályokkal ellentétben nem hozhatunk létre modulra épülő objektumot.

A moduloknak két fő felhasználási területe van, az egyik hogy névtérként viselkednek, amelyben definiált metódusok nevei nem ütköznek a más helyen lévő metódusok neveivel, másodsorban funkcionalitást lehet megosztani osztályok között melyek egy modulban vannak.

A Rails sűrűn használja a modulokat, a segítő metódusok is modulokban vannak. Egy segítő modult automatikusan létrehoz minden nézet sablonhoz.

## Osztályok

Az osztályok a ***class*** kulcsszóval kezdődnek amelyet az osztály neve követ, ezután lehet megadni hogy melyik osztály alosztálya. Érdemes odafigyelni hogy a konstruktorok meghívása a ***new*** metódussal történik, meghívásuk az osztályon belül az ***initialize*** metódussal lehetséges. Öröklődést, származtatást a < jellel definiálhatunk.

```
class OszalyNev
end
```

## Hozzáférési módosítók

Metódusok kaphatnak, alapesetben ezek ***public*** módosítóval rendelkeznek tehát mindenki hozzáférhet az adott metódushoz. Lehet ***protected*** illetve ***private*** módosítóval ellátni a metódusokat, előző esetén az adott példány vagy ugyanaz az osztály vagy alosztálya hívhatja meg, privátnál csak az adott példány hívhatja meg. Különbség más nyelvekhez képest hogy a módosítók a hatásukat nem az adott metódusra fejtik ki hanem területre érvényesek és addig érvényesek amíg más módosítót nem alkalmazunk.

## Metódusok

A metódusok ***def*** kulcsszóval kezdődnek majd a metódus neve jön végül a paraméterek listája zárójelek között, a metódus törzsét nem határolja {} jel, a metódus a paraméterlistát záró zárójeltől az ***end*** kulcsszóig tart.

A metódus visszatérési értéke az utolsó kiértékelés vagy a ***return*** kulcsszó hatására visszaadott érték.

```
def koszontes(nev)
  eredmeny = "Üdvözöllek, " + nev
```

```
  return eredmeny
end
```

A sorok végre nem szükséges pontosvessző ha minden utasítás külön sorba kerül. A # jellel kezdődő sorok  megjegyzéseknek minősülnek.

## Attribútumok

Az osztály törzsén belül definiálhatunk osztályszintű vagy példányszintű metódusokat. Ha egy metódus „self.”-el kezdődik akkor az osztályszintű metódus.

Az adattagok az osztályon belül @ jellel kezdődnek, ezek a felelősek az osztályon belüli állapot tárolásáért. Kívülről nem érhetők el közvetlenül, csak metódusokon keresztül.

## Kivételkezelés

A kivételek objektumok, ***Exception*** osztály leszármazottja vagy az alosztályai tagja. A ***raise*** metódus váltja ki a kivételt, ami megszakítja a program normál futását, mely közben a Ruby megvizsgálja a hívás listát hol van az adott kivétel lekezelve. A ***rescue*** kulcsszóval tudjuk az adott kivétel elkapni.

```
begin
    #utasítások, ahol a kivétel kiváltódhat
    rescue
    #kivételkezelés helye
end

def fajlba_mentes
    begin
    File.open ('mentes.txt','w') { |file|
        file.puts @adat
            file.puts @adat2}
    rescue Exception => ex
    $stderr.print"A fájl megnyitása sikertelen"
    end
end
```

## 3.4 Vezérlés és elnevezési konvenciók

## Vezérlési struktúrák

A Ruby rendelkezik az összes vezérlési struktúrával mint a többi modern programozási nyelv, elágaztató utasítása az ***if***, ciklus utasítása a ***while***. Nincsenek kapcsos zárójelek, a struktúrák végét egy ***end***-el jelezzük.

```
if benzin_szazalek < 0.85
  puts "Sok benzin van."
elsif  benzin_szazalek < 0.85 and  benzin_szazalek > 0.3
  puts "Van elég benzin"
else
  puts "Fogy a benzin, tankoljon!"
end
```

*While ciklus:*

```
while benzin_szazalek < 100 and penz > 0
  benzin_szazalek += 0.01
  penz -= 1
end
```

## Ruby nevesített konstansok

Lokális változók, metódus nevek, metódus változók nevek kis betűvel vagy aláhúzással kell hogy kezdődjenek. A példányosított változók „@” jellel kezdődnek. A Ruby elnevezési konvenciója hogy a több tagú változóneveket aláhúzással különítsük el. Az osztályok, modulok, konstansok nagy betűvel kell hogy kezdődjenek. Itt a több tagú neveket nagybetűkkel választjuk és nem aláhúzással.

A Rails nagy hangsúlyt fektet a szimbólumok használatára, a szimbólumok hasonlóak a változókhoz csak a nevük kettősponttal kezdődik. A szimbólumokat dolgok azonosítására szolgál, például metódusok paraméterei.

### 3.4.1 Blokkok és iterátorok

A blokkok nem mások mint kódrészletek {} vagy ***do*** és ***end*** között. Rails esetében a konvenció hogy egysoros blokknál a kapcsos zárójel a használandó, a hosszabbaknak a do, end páros.

```
{ puts 2+2 }          # Ez egy blokk
do
   szam1=2
   szam2=2
   eredmeny = szam1+szam2
end
```

A blokkok számos esetben az iterátorokkal együtt jelennek meg. Az iterátorok olyan metódusok melyek segítségével kollekciók egymás utáni elemeihez tudunk hozzáférni, mint például a tömb. Ez a metódus az each, példa a használatára.

```
tomb = [''cica'', 2, 13,8]

tomb.each {|elem| puts elem}
```

## Elnevezési konvenciók

A Rubyban a változó nevek kisbetűsek és ahol el kell választani őket ott aláhúzással tesszük meg. Az osztályoknál és a moduloknál nincs aláhúzás és az összetett szavaknál a tagok nagy betűvel kezdődnek.

A Rails estében ez azért fontos mert ezek szerint generál automatikus neveket. Feltételezi hogy az adatbázisok tábla nevei kisbetűsek mint a változó nevek, a fájl nevek is kisbetűsek aláhúzással elválasztva és hogy a tábla nevek többesszámban vannak.

## 3.5 Ruby segédprogramok

## IRB – Iteraktív Ruby Shell

Az IRB abban segít hogy ne kelljen fájlokat írnunk ha valami újat akarunk

kipróbálni. Meghívása történhet parancssorból vagy IDE által, az IRB minden soremelés leütése után értelmezi a kódot, végrehajtja és kiírja a kimenetre a művelet eredményét. Elmenti a munkamenetben kiadott parancsokat, így újra tudjuk futtatni őket így nem kell őket újra begépelni.

## RAKE

A RAKE létrehozásának a célja hogy a GNU make-hez hasonló forráskód kezelő programot hozzanak létre. A Ruby programok futtatásához nincs szükség futtatható állomány generálására, a RAKE inkább telepítésre valamint gyakori feladatok rögzítésére majd elvégzése a cél, ezeket a Rubyban megírt kódokat taszkoknak hívjuk, leggyakrabban adatbázis-műveletekhez használják.

## RubyGems

A Ruby saját csomagkezelője, a függőségeket figyelembe véve könyvtárakat, programokat telepíthetünk, törölhetünk, hozhatunk létre. Ezek során dokumentációt is előállítja, ezzel a Rubygems az egyik legpraktikusabb és legsokoldalúbb Ruby eszköz, hiszen lényegesen megkönnyíti dolgunkat.

# 4. Ruby on Rails részletesen

## 4.1 A Ruby on Rails eredete

David Heinemeier Hansson fejlesztette ki a keretrendszert a BaseCamp nevű projekt fejlesztése közben. A fejlesztő először 2004 júliusában adta ki az első változatot nyílt forrással, MIT licencel. Az igazi fordulópont akkor történt amikor az Apple bejelentette hogy a Mac OS X v10.5 változatát a Ruby on Rails keretrendszerrel együtt szállítják. Ez nem csak a Rails szempontjából volt fontos hanem a Ruby mint programozási nyelvnek is, mely legfőképp a Rails miatt terjedt el.

"Rails is the killer app for Ruby."
*Yukihiro Matsumoto, A Ruby készítője*

A keretrendszer fejlesztését egy többtagú, úgynevezett core team végzi, amelynek vezetője a David Heinemeier Hansson dán programozó. Ebben a pillanatban elérhető legfrissebb stabil verzió a 2.3-es verziószámot viseli.

A Rails sok újszerű meglátást, olyan programozástechnikai elveket és módszereket hozott a webalkalmazások készítésére mint a DRY, COC, Agilis fejlesztés, Egység Tesztelés stb., ami eddig elég ritkán alkalmazott módszerek voltak.

## 4.2 Naplózás és dokumentálás

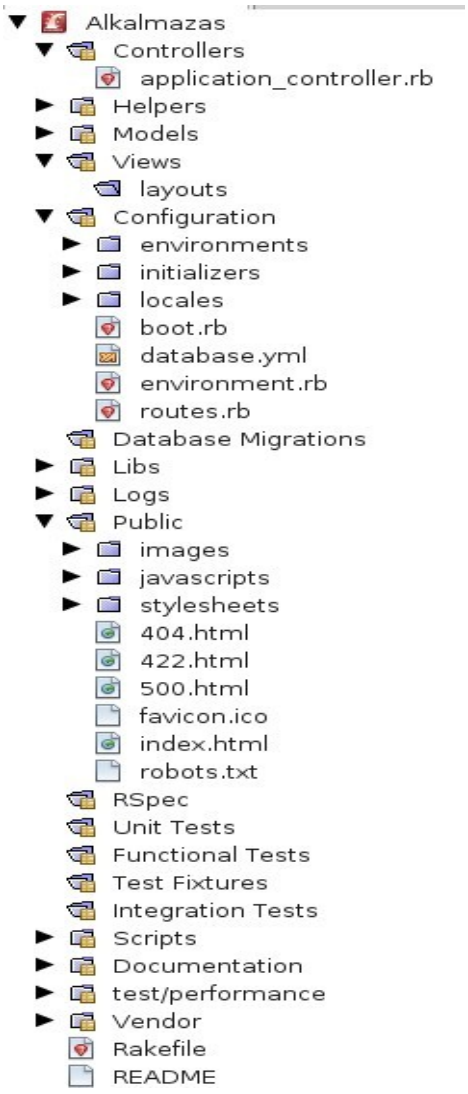
A naplózási funkciót eleve tartalmazza a Rails, a ***`Logger`*** osztály végzi ami bármelyik osztályból meghívható. Négy szint alapján lehet a naplózandó eseményeket osztályozni, ez sorrendbe a ***`warn, info, error, fatal`***. Be lehet állítani hogy mely szinttől írja be az események a naplózó. A naplózási fájlok a log könyvtárban vannak, az események mentési helye függ a futási környezettől, ha éppen teszteljük az alkalmazást akkor az ***`log/test.log`*** lesz a napló fájl neve.

Az RDoc a Rails beépített dokumentálási rendszere ami forrásfájlból készít HTML dokumentációt. A forrást végignézi és osztályokat, metódusokat, modulokat és a

hozzátartozó megjegyzéseket és csinál egy strukturált dokumentációt. A ***rake doc:app*** parancs hatására dokumentációt a ***doc/app*** könyvtárban helyezi el.

## 4.3 Rails könyvtár struktúra

Jól megtervezett könyvtárszerkezetet használ a Rails. A vezérlő, a nézet és modell fájlai mind az ***app*** könyvtárba kerülnek mentésre. A legfelső szinten van egy Rakefile ami tesztek, dokumentációk készítésére használhatunk. Az ***app*** és a ***test*** könyvtárban lesz az alkalmazás magja.




*ábra 2: A Rails könyvtárstruktúrája*

contollers: A vezérlők helye a app/controllers mappában van, minden vezérlőhöz

tartozik egy controller_neve.rb fájl illetve itt van még a application.rb is ami minden vezérlő ősosztályát tartalmazza, ez az ApplicationController.

helpers: A vezérlőhöz tartozó segítő fájl helye a app/helpers mappában van, minden vezérlőhöz tartozik egy helper_neve.rb fájl illetve itt van még a application _helper.rb is amiben a ApplicationHelper modul van benne, minden vezérlő számára elérhető.

views: A nézetek helye a app/helpers mappában van, minden vezérlőhöz új mappa jön létre, ezekben lesznek a vezérlőhöz tartozó nézetek. A layout mappában pedig a vezérlőhöz tartozó elrendezések.

config: A config könyvtárban találhatók a beállítások.

boot.rb: indítási parancsok (ezt elvileg nem ajánlott módosítani).

environment.rb: környezeti beállítások, pl. milyen Rails verziót használunk, vagy mik a session beállításai.

routes.rb: az útvonalak beállításai, hogy milyen címhez milyen vezérlő milyen eseményét kell meghívni, milyen paraméterekkel.

database.yml: megadja YAML formátumban, hogy milyen paraméterekkel lehet elérni a különböző módok adatbázisait.

environments: Egy Rails alkalmazás háromféle módban működhet (ettől függően háromféle beállításhalmazt lehet definiálni, és mindegyikhez külön adatbázis tartozik):

development: fejlesztői mód, jellemzően cache nélkül, debug elérhetőséggel.

production: normál üzemmód, cache-sel (emiatt ha változtatás van, akkor újra kell indítani az alkalmazás szerverét)

test: teszt üzemmód, adatbázisát minden egyes alkalommal lenullázza és újra generálja (ezért ezt mindenképp el kell különíteni)

initializers: Különböző feladatokat lehet az ebben a könyvtárban lévő fájlokkal végrehajtani a szerver felállásakor.

db: Az adatbázissal kapcsolatos fájlok találhatók itt, a migrate könyvtárban az egyes migrációs lépések, a schema.rb fájlban pedig az aktuális séma.

lib :Egyéni fájlokat lehet ide rakni, amik bárhonnan elérhetők, így pl. az application.rb-be beimportálva (require) egy ilyen fájlt akár elrejtett beállításokat is felül lehet definiálni.

log: logok tartoznak ide, minden módhoz külön fájlban.

public: Ebben a könyvtárban indul az alkalmazás futása. A szerver típusától függően valamelyik dispatch fájl fog elindulni. Emiatt itt vannak olyan fájlok, mint a hibaoldalak,

favicon.ico, index.html.

images: A felhasznált képeket ebbe a könyvtárba kell elhelyezni, egy adott kép eléréséhez helper-ek használhatók.

javascripts: Itt találhatók az Ajax-hoz szükséges és más JavaScript fájlok, valamint a saját JavaScript-jeink is az application.js fájlban. Az alap JavaScript fájlokat egy template (pl. layout) head-jében a javascript_include_tag :defaults paranccsal illeszthetjük be.

stylesheets: Minden vezérlőhöz külön stíluslap tartozhat ebben a könyvtárban. Ezeket a stylesheet_link_tag 'nev' paranccsal illeszthetjük be egy template-be.

script: Itt található pár Rails-hez kapcsolódó Ruby scriptek pl. generátorok végrehajtani.

test: A tesztekhez szükséges fájlok találhatók itt.

tmp: Itt található a cache, a session fájljai (amennyiben fájlokban tároljuk a felhasználók session adatait), és mások. Ezeket ajánlatos törölni ha új gépre visszük át az alkalmazást, vagy valami hiba történt.

vendor: Ha föl van telepítve a Subversion (svn) a gépünkre, akkor ebbe a könyvtárba csak ehhez az alkalmazáshoz föltelepíthetünk egyedi Rails verziókat.

## 4.4 Migrációk

A Rails előnyben részesíti az agilis megközelítést, így az adatbázis műveleteknél is. Az adatbázis műveleteket nyilván tartja mint például a tábla létrehozást, oszlopok átnevezése. Ezeket és általában a DDL műveleteket tároljuk, ellátjuk verziószámmal ami az UTC szerinti aktuális dátum és idő.

A verziószám kezelést a schema_migrations táblázat alapján készíti ami aminek csak egyetlen oszlopa van verziószám. Ha az aktuális állapotra szeretnénk frissíteni az adatbázis akkor először megvizsgálja hogy van-e több migráció amit nem hajtottunk végre, ha van akkor ezeket időrendi sorrendben végrehajtja egyenként. Az adatbázis együtt fejlődik az alkalmazással.

Az migrációkat vissza is lehet vonni. A migrációk két utasítást tartalmaznak, az egyikkel a változásokat amit szeretnénk az adatbázison alkalmazni. A másik utasítással visszaállítható a migráció alkalmazása előtti állapot. Ezt úgy réi el hogy megnézi melyik migráció verzióállapotot akarjuk visszaállítani, ha az aktuális verziószám nagyobb akkor

azt a migrációt visszavonja, ezt addig ismétli amíg el nem éri a kívánt adatbázis verziót.

A migrációk egyszerű Ruby fájlok a ***db/migrate*** könyvtárban. Minden migrációs fájl egy számsorozattal és egy aláhúzással kezdődik, ez a számsorozat a migráció verziószáma. A migrációs fájlokat magunk és létrehozhatjuk de van rá automatikus eszköz is. Két módon hozható létre. Az egyik ha modellt hozunk létre ezzel egy új migrációs fájl is létrejön, aminek a neve a ***verziószám_modellnév.rb*** lesz, de létre hozhatunk saját magunk is léte hozhatunk migrációs fájlt.

```
ruby script/generate model discount
ruby script/generate migration add_price_column
```

### 4.4.1 Migrációk felépítése

A migrációk az ActiveRecord::Migration osztály alosztálya, aminek két metódusa van az ***up*** és a ***down***. Az ***up*** metódus ami a migráció futtatása közbeni változtatásokat tartalmazza, a ***down*** metódus a migráció visszavonási parancsait gyűjti össze.

```
class SomeMeaningfulName < ActiveRecord::Migration
  def self.up
    #törzs
  end
  def self.down
    #törzs
  end
end
```

Az adatbázis migrációval sok lehetőséget ad, mivel a Rails az migrációkat adatbázistól függetlenül kezeli, de az adatbázisokat adaptereken keresztül éri el, így nem biztos hogy az összes funkció elérhető az összes adatbázissal.

Az attribútumok típusai a Rubyban létező típusok lehetnek, ezeket az adapter az adott adatbázis legközelebbi típusát alkalmazza ami biztosan tudja tárolni a Ruby típusok tartományát.

| | db2 | mysql | oracle | postgresql | sqllite |
|---|---|---|---|---|---|
| :binary | blob(32768) | blob | blob | bytea | blob |
| :boolean | decimal(1) | tinyint(1) | number(1) | boolean | boolean |
| :date | date | date | date | date | date |
| :datetime | timestamp | datetime | date | timestamp | datetime |
| :decimal | decimal | decimal | decimal | decimal | decimal |
| :float | float | float | number | float | float |
| :integer | int | int(11) | number(38) | integer | integer |
| :string | varchar(255) | varchar(255) | varchar2(255) | character varying(256) | varchar(255) |
| :text | clob(32768) | text | clob | text | text |
| :time | time | time | date | time | datetime |
| :timestamp | timestamp | datetime | date | timestamp | datetime |

Az adatbázis migrációkkal lehetséges új oszlopokat beszúrni, átnevezni az oszlopokat és típusokat megváltoztatni, új oszlopot az ***add_column*** metódussal lehet beilleszteni. A példa egy string típusú attribútumot illeszt be az első paraméterül megadott táblába.

```
class AddEmailToOrders < ActiveRecord::Migration
  def self.up
    add_column :orders, :e_mail, :string
  end

  def self.down
    remove_column :orders, :e_mail
  end
end
```

Az alkalmazás fejlesztése közben gyakran változhat az adatbázis terv, például ha egy oszlopnak egy jobb nevet találunk. Az oszlopok átnevezése a ***rename_column*** metódussal lehetséges:

```
class RenameEmailColumn < ActiveRecord::Migration
  def self.up
    rename_column :orders, :e_mail, :customer_email
  end
  def self.down
    rename_column :orders, :customer_email, :e_mail
  end
end
```

Az átnevezés nem törli az oszlophoz tartozó adatokat, illetve néhány adatbázis adapter nem is támogatja, oszlopok típusának megváltoztatása is lehetséges, ezt leginkább számból stringgé alakításnál használatos. Ilyenkor a konverzió egyszerű, az átalakítás gond nélkül végrehajtható. A másik irányból azonban nem ilyen egyszerű a változtatás, az ilyen esetekre van egy külön kivétel. Amit a migrációk visszavonásánál hasznos hogy ne tudjuk végrehajtani. Az oszlopok típusának megváltoztatása a ***change_column***.

```
class ChangeOrderTypeToString < ActiveRecord::Migration
   def self.up
     change_column :orders, :order_type, :string, :null =>
      false
   end
   def self.down
     raise ActiveRecord::IrreversibleMigration
   end
end
```

### 4.4.2 Adat migrációk

Az adatbázis szerkezeti szerkezeti változásain kívül lehetséges tartalmi változást is elérni migrációk segítségével. Ez legfőképp tesztelésnél fontos hiszen a szerkezeti változtatások érdemben akkor látszódnak ha adatok is szerepelnek az adatbázisban, a másik eset amikor fix adatokkal kell feltöltenünk az adatbázist. Az adatokat lehet fájlból vagy migráción belül is meg lehet adni.

```
class DataUsers < ActiveRecord::Migration
```

```
  def self.up
    down
    user = user.create(:name => "Teszt Elek" ,
                               :age => "23" ,
                               :money => "1500" )
    user = user.create(:name => "Tóth Béla" ,
                               :age => "18" ,
                               :money => "12200" )
    user.save!
  end
  def self.down
    User.delete_all
  end
end
```

## 4.5 Active Record

Az Active Record a Rails ORM megvalósítása, ez teszi lehetővé hogy az adatbázishoz kapcsolódjunk, a táblákat osztályokkal feleltessük meg és hogy az adatokat meg tudjuk változtatni.

Az Active Record ORM implementációja az alapértelmezett módon kezeli az osztályok, objektumok viszonyát az adatbázissal. A táblákat az osztályokhoz, a sorokat az objektumokhoz, az oszlopokat pedig a osztály attribútumaihoz rendeli. Amiben eltérés van a többi ORM-et megvalósító eszközzel az az hogy alapértelmezetten sok paraméter meg van adva így nem kell sok időt tölteni a konfigurációra.

A táblanevek és az osztálynevek alapértelmezetten úgy lesznek létrehozva hogy az osztályok egyes számban, a tábla nevek pedig többes számban vannak, elválasztás esetén az osztálynevek nagybetűvel emeljük ki a tagok kezdőbetűit a táblaneveknél aláhúzással tagolunk.

Ezek legfőképp a Rails megalkotójának a dán David Heinemeier Hansson-nak a filozófiáját tükrözi, ha esetleg ez nem tetszik vagy eleve megadott táblával kell dolgoznunk, lehetőség van ezt felülírni. Ezt két metódussal tudjuk megtenni.

```
class Country < ActiveRecord::Base
   set_table_name "country_old_table"
```

```
end
```

illetve:

```
class Computer < ActiveRecord::Base
   self.table_name = "ITStuff"
end
```

Az Active Record az objektumokat a táblák sorainak felelteti meg. Az objektumok adattagjait pedig a táblák oszlopai reprezentálja. Az osztályon belül nem kell extra információt megadni hogy a táblák oszlopai melyik adattaghoz tartoznak, mivel a Rails ezt dinamikusan kezeli.

Ha egy adatbázis táblát hozunk létre egy osztályra alapozva akkor automatikusan három oszlopa eleve létre lesz hozva, ezek az ***id, created_at, modified_at***. Az ***id*** oszlop lesz az alapértelmezett elsődleges kulcs, pozitív egész szám, egyedi és minden egyes új sor hozzáadásával növekszik egyel az értéke az előzőhöz képest. A ***created_at*** és ***modified_at*** timestamp típusú és nevükből adódóan a sor keletkezését és módosítási időpontját jelölik. Amennyiben egy létező adatbázist kell felhasználnunk vagy más oszlopot akarunk használni akkor mód van arra hogy elsődleges kulcs attribútumot megváltoztassuk.

```
class Auto < ActiveRecord::Base
  self.primary_key = "alvaz_szam"
end
```

Az Active Record nem támogatja a több részből álló elsődleges kulcsot, sem a létrehozásukat, sem a felhasználásukat, viszont létezhet olyan plugin amellyel ez lehetővé válik.

Az Active Record az adatbázissal a kapcsolatot adapterekkel oldja meg, megpróbálja a lehető legmagasabb absztrakcióval kezelni, amivel az adatbázisok cseréjét könnyíti meg, ez viszont csorbulhat amikor natív sql parancsokat alkalmazunk a kódban. Lehetőség van saját adapter definiálni az ***establish_connection*** osztály segítségével.

```
ActiveRecord::Base.establish_connection(
```

```
  :adapter => "sqlite3" ,
 :database => "db/railsdb.sqlite3" )
```

Az adatbázis típusa és konfigurálása a config/database.yml fájlban történik, ami az alapértelmezett és az ajánlott, hogy a kód elkülönülve maradjon az adatbázis specifikációtól.

A natív SQL kód használata elkerülése végett az Active Record sok metódust tartalmaz amivel a leggyakoribb adatbázis műveletek hajtatók végre mint a adatok felvitele, lekérdezése, módosítása.

A legegyszerűbben egy objektumot úgy tudunk lementeni ha példányosítjuk, megadjuk az adattagoknak a kívánt értékeket és az osztály ***save*** metódusával az objektum bekerül az adatbázisba.

```
client = User.new
client.name = ''Tóth Béla''
client.email = "toth.bela@clientcompany.com"
client.save
```

Ezt le lehet rövidíteni a ***create*** metódussal ami egyben példányosít és menti az objektumot az adatbázisba.

```
an_order = User.create(
  :name => "Tóth Béla" ,
 :email => "toth.bela@clientcompany.com" )
```

Adatot az táblákból több módon meg lehet tenni, minden modell osztály támogatja a ***find*** metódust, ami alapesetben a tábla elsődleges kulcsa alapján keresi a táblából a keresett adatokat. Ha létezik az adott paraméterű sor akkor az objektummal tér vissza amit talált a táblában lévő adatokkal tölt fel, ellenkező esetben kivétel váltódik ki, a ***RecordNotFound***.

```
client = User.find(14)
```

Lehetőség van a ***find*** metódus bővített használatára, több paraméterrel ellátni. Az ***:all*** paraméterrel az összes találatot kilistázza, az ***:order***-el rendezni lehet, a ***:limit*** a leszűkíti a találatokat az első ***:limit*** által megadott értékre. Ezeken kívül még sok

lehetőség van, mellyel a kereséseket finomhangolni lehet.

```
clients = User.find(:all,
                    :conditions => "name = 'Béla'" ,
                    :order => "age DESC" ,
                    :limit      => 10)
```

Ez egyik érdekes lehetőség a lekérdezéseknél a dinamikus metódus használatára, ha a tábla egy attribútumára akarunk keresni akkor az Active Record futási időben legenerálja a ***find_by_ , find_last_by_ , find_all_by_*** metódusokat amit kiegészül az attribútum nevével. Ezek értelemszerű funkció látnak el, attribútum szerint kereshetünk egy, az utolsó vagy összes eredményt, ezen parancsoknál szintén használhatók a ***find*** paraméterei.

```
client = User.find_by_name("Tóth Béla" )
clients = User.find_all_by_age("28" )
```

A parancs utáni felkiáltójel segítségével kivételt (ActiveRecord:: RecordNotFound ) válthatunk ki, ha a keresés nem adott vissza eredményt ha a NIL nem megfelelő nekünk.

```
client = User.find_by_name!("Ilyennincs Béla" )
```

Az adatok törlése a ***delete*** és a ***delete_all*** paranccsal történik ami adatbázis szinten hajtódnak végre, ez a parancs az adapteren keresztül hajtja végre a műveletet.

```
User.delete(123)
User.delete_all(["age < ?" , @minimum_age])
```

A másik mód a ***destroy, destroy_all*** metódus, ami az Active Recordon keresztül hajtja végre a törlést. A különbség a két metódus között hogy a ***destroy*** leállítja az adott objektummal kapcsolatos összes műveletet hogy az adott objektum, de az összes validálási és más ellenőrző metódusok megmaradnak, ezzel konzisztens marad az adatbázis. A ***delete*** azonnal végrehajtja a törlést, nem biztosítja a konzisztenciát, a ***destroy*** az ajánlott.

## 4.6 Táblák közötti kapcsolat

A gyakorlatban sok a több táblás alkalmazás és fontos a közöttük lévő kapcsolat megteremtése és fenntartása. Adatbázis szinten ezek külső kulcsokkal érhető el mint hivatkozás. Ez elég alacsony szintű mivel sorokat és táblákat kell kezelnünk és nem felhasználó barát, sokkal kényelmesebb az ORM megközelítés amit a Rails is biztosít.

Ezzel az alacsony szintű megvalósítással nem kell foglalkozni, mindig objektumokat és objektumok közötti kapcsolatot látunk és kezelünk. Itt a is a külső kulcs lesz alapja a kapcsolatnak, amit nekünk kell definiálni az adatbázis séma leírásánál, ezek az oszlopok integer típusúak és a nevük pedig a tábla neve _id végződéssel. Az Active Record három féle kapcsolatot ismer és kezel.

### 1:1 kapcsolat

Ez a kapcsolattípus akkor áll fenn ha a külső kulcs szerepel egy másik táblában és résztvevők egymással kölcsönösen megfeleltethetők, vagyis a a résztvevők mindkét részéről csak 1 entitás szerepel. Ilyen lehet az autó-rendszám kapcsolat. Ezt a ***has_one*** és ***belongs_to*** metódusok modellben való feltüntetésével lehet elérni, fontos megjegyezni hogy mindig abban a modellben van a ***belongs_to*** ahol a külső kulcs szerepel.

### 1:N kapcsolat

Ezzel egy objektumhoz egy objektumok kollekcióját lehet rendelni. Ilyen az anya – gyermek kapcsolat. Ezt a ***has_many*** és ***belongs_to*** metódusok modellben való feltüntetésével lehet elérni, a ***has_many*** a kollekciót hivatkozó elemnél, a ***belongs_to*** most is a külső kulcsot tartalmazó modellek leírásában van.

### N:N kapcsolat

A több - több kapcsolatot két módon valósítható meg. Az egyik ha kapcsolótábla segítségével jön létre a kapcsolat ami csak a kapcsolatban résztvevő táblák kulcsait

tartalmazza. Ezt a ***has_and_belongs_to_many*** metódus modellben való leírásával jelezzük. A Rails feltételezi hogy a kapcsolótábla neve a két tábla neve ABC sorrendben aláhúzással elválasztva.

A másik mód ha szükségünk van a köztes táblára, mint modell, hogy ellenőrzéseket társítsunk a rekordokhoz mentés előtt akkor a ***has_many...*** és ***:through => …*** asszociációt kell használunk. Itt a ***:through*** opcióval közvetetten kapcsolódunk a köztes, úgynevezett kapcsolat modellhez, amely összeköti a másik két hozzá kapcsolódó modellt.

A kapcsolatok megadása után a Rails dinamikusan hozzáad a modell osztályához metódusokat, amivel megadhatjuk és elérhetjük a hivatkozott modelleket.

## 4.7 Validáció

Az Active Record segítségével tudjuk a modell adatait validálni, ez a validáció történhet automatikusan megtörténik az objektum mentésekor, ha ez nem sikerült akkor nem kerül mentésre az objektum. A validálásnál különbséget tehetünk az új és a létező adatok között erre a ***new_record*** metódus használható, ami true értékkel tér vissza ha új rekordról van szó ezt a ***validate_on_create, validate_on_ update*** metódusokkal is lehet ellenőrizni.

- validates_presence_of

  Az adott attribútumot kötelezően meg kell adni.
- validates_numericality_of

  Az adott attribumnak szám típusúnak kell lennie
- validates_acceptance_of

  Ellenőrzi hogy a paraméterként megadott jelölőnégyzet be van-e jelölve.
- validates_confirmation_of

  Olyan adatok ellenőrzését végzi amelyeket többször adunk meg és azok azonosságát vizsgálja.
- validates_length_of

  A megadott attribútum hosszúsgát lehet ellenőrizni ezzel.
- validates_uniqueness_of :attributum, :scope => ...

  Az attribútum nem vehet fel a táblában már létező értéket, a scope opcióval több

attribútumot is figyelembe vehetünk.

- validates_inclusion_of :attributum, :in=>(felsorolas)
  Ellenőrzi hogy az attribútum értéke szerepel-e a felsorolásban.
- validates_exclusion_of :attributum, :in=>(felsorolas)
  Ellenőrzi hogy az attribútum értéke nincs-e a felsorolásban.
- validates_format_of :attributum, :with=>/reguláris kifejezés/
  Összeveti az attribútumot a reguláris kifejezéssel.

Lehetséges a validációt feltételhez kötni. Ezt az ***:if*** és az ***:unless*** segítségével lehet végrehajtani, amennyiben a feltételnek nem felelnek meg a validáció nem kerül végrehajtásra. Ez akkor jöhet jól ha egy validáció feltétele egy bizonyos eshetőség, például ha a jelszó és a jelszó megerősítés mezőt akarjuk összehasonlítani, aminek értelemszerűen feltétele hogy jelszó mező ne legyen üres.

```
validates_confirmation_of :password,
  :message => "Egyeznie kell a jelszóknak" ,
  :if => Proc.new { |i| !i.password.blank? }
```

Az Action Pack tartalmazza az Action Contollert és Action View-t, ez az agya az alkalmazásnak, a bejövő kéréseket irányítja és  a válaszokat is az Action Pack állítja elő.

Az Action Controller felelős az útvonalakért, amihez két módot ad meg az egyik a kényelmesebb előre felállított, amikor alapvető szabályokat adunk meg és mintaillesztéssel feleltetjük meg ezt az beérkező kérésekre. A másik a hagyományos módszer amikor kézzel állítjuk be az útvonalakat. Mindkét módszer támogatott és szabadon keverhető.

Alapesetben amikor létrehozunk egy új Rails alkalmazást, felépül a könyvtárstruktúra és az alap konfigurációs beállítások. A ***config/routes.rb*** tartalma a következő:

```
ActionController::Routing::Routes.draw do |map|
  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'
end
```

A Routing objektum állítja be az alkalmazásunk hivatkozási térképét amivel URI-kel tudjuk hivatkozni alkalmazásunk erőforrásait. A ***map.connect*** hatására kerül be cím a térképbe, ezután jön a '*:controller/:action/:id*' ami egy mintaillesztés, ekkor az összes URI-t ami három tagból megfelelteti a mintaillesztéshez és keres egyezést. Amennyiben a keresés nem járt sikerrel akkor hibaüzenetet kapunk vissza. Az alapeseti mintaillesztés '*vezerlo/akcio/:id*' formában értelmezi a bejövő kéréseket, például a *http://railsalkalmazas.hu/users/edit/2* kérés a ***UserController*** osztály ***edit*** metódusát hívja meg ahol az :id 2-vel lesz egyenlő.

## 4.8 Action Controller

Amikor egy kérést kezel a vezérlő akkor megnézi hogy létezik-e publikus metódus aminek a neve megegyezik a bejövő kéréssel, ha van akkor a metódus meghívódik. Ellenkező esetben vezérlő egy ***method_missing*** nevű metódust generál, majd keres egy sablont ami megegyezik a bejövő kérés nevével ha az sincs akkor ***Unknown Action*** hibaüzenetet kapjuk.

Minden a vezérlőben lévő metódus elérhető kívülről, ennek kikerülése végett lehet a metódusokat ellátni ***protected*** vagy ***private*** láthatósági módosítóval, ha szeretnénk hogy a metódusaink elérhetőek legyenek a többi osztály részére de kívülről nem akarjuk hogy hivatkozzák akkor ***hide_action*** metódussal lehet elrejteni.

Hozzá lehet férni a kérés részletes adataihoz amit a Rails objektumokban tárol ilyenek a ***action_name, cookies, headers, params, request, response*** és a ***session***. A vezérlő egy kérésre csak egy választ adhat így erre figyelnünk kell különben a ***DoubleRenderError*** kivétel váltódik ki. Ezt a ***redirect_to, render*** és a ***send_***... metódusok többszörös meghívása válthatja ki.

A ***render*** segítségével az alapértelmezettől eltérő nézetet definiálhatunk, alapesetben a *vezérlő/esemény.html.erb* a nézet fájl. Ezzel a metódussal átirányítható a egy másik vezérlő nézetére, de ekkor nem fog lefutni az eredeti esemény metódusa, erre a ***redirect_to*** metódus szolgál. A ***send_data*** és a ***send_file*** segítségével stringtől eltérő adatokat küldhetünk az alkalmazás felé, az első metódus tágabb funkcióval

rendelkezik, adatfolyamot küld, a második metódus pedig konkrétan fájl küldésre használható.

### 4.8.1 Szűrők

A vezérlőben elhelyezhetünk szűrőket amelyekkel időzíthetjük metódusaink futását, ez hasznos olyan funkcióknál mint a autentikáció, naplózás, stb. A Rails három fajta szűrőt támogat, ezek a ***before_filter, around_filter*** és az ***after_filter***.

A ***before_filter*** egy esemény előtt hívódik meg, mielőtt a vezérlőben bármi is történne először az összes ***before_filter*** hajtódik végre és csak utána a vezérlőben lévő utasítások. Az ***after_filter*** szűrő egy esemény után hajtódik végre, a vezérlő utasítási hajtódnak végre és utána ez a fajta szűrő.

```
class AdminController < ApplicationController
  before_filter :authorize, :only => [ :delete,
  :edit_comment ]
  after_filter  :log_access, :except => :rss
```

A szűrőket két fajta opcióval lehet ellátni az egyik az *:only*, ami a szűrőt csak akkor hajtja végre ha a paraméterként megadott eseménnyel hivatkozott a vezérlőre, az *:except* opció hatására csak akkor nem lesz aktív a szűrő ha a paraméterként megadott esemény következik be.

Az ***around_filter*** speciális a maga formájában hiszen egyszerre lehet vele ***before_filter*** és ***after_filter*** hatásait elérni. A szűrő kódja az esemény végrehajtása előtt megkapja vezérlést, és végrehajtja a szűrő kódjába lévő ***yield*** utasításig, ekkor az esemény fut le és amint az esemény végrehajtódott folytatódik a szűrő a ***yield*** utáni parancsokkal. Amennyiben nem tartalmaz a szűrő ***yield*** utasítást a vezérlés soha nem kerül az eseményhez. Ez a szűrő típus főleg akkor hasznos ha a vezérlőben lévő utasítások fontos a szűrő működése szempontjából.

## 4.9 Action View

Az Action View már a nevén is látszódik hogy az MVC hármasból a nézetet valósítja meg. Az összes funkcionalitást, technológiát gyűjti össze amivel az alkalmazás nézeteit tudjuk létrehozni, általában HTML, XML, JavaScript kódot generálva. Alapértelmezetten az ***app/view*** konyvtárban helyezkednek ahol minden egyes vezérlő részére külön alkönyvtár jön létre, ettől el lehet térni erre a ***render*** metódus ad lehetőséget.

A sablonok keverve tartalmaznak statikus szöveget és kódot amivel a dinamikus részét adhatjuk meg az oldalnak, a kód rész **<% %>** jelekkel van határolva. A sablonokban minden példány szintű változó elérhető a vezérlőkből, ezzel biztosítva a kommunikációt. A vezérlő által biztosított objektumok mint a ***flash, params, request, params***, stb. is elérhető a nézetből, de a ***flash*** kivételével nem ajánlott ezeket közvetlenül használni mivel ezek az objektumokat a vezérlő hatásköre, a ***controller*** nevű objektum segítségével tudjuk az aktuális vezérlő által kínált metódusokat meghívni.

A Rails Action View három fajta sablont ismer és kezel az elsővel az XML válaszokat lehet készíteni ehhez a Builder osztállyal. A második ERb sablonok amiben a kód és statikus szöveg keveredik, ez végül HTML formára alakítja. Az RJS sablonokkal Javascript kódokat tudunk futtatni, ez általában az Ajax-os oldalakon található meg.

### 4.9.1 Helper

A Helper metódusok a nézetet felesleges kódoktól szabadítja meg úgy hogy azokat külön választja, ez is az MVC szemléletet biztosítja hogy minél kevesebb üzleti logika legyen a nézetben. A ***helper*** egy modul ami nem más mint metódusok összessége ami a nézetek rendelkezésre áll, a különválasztás a tesztelés miatt is hasznos hiszen nem egy alapvetően HTML fájlt, hanem egy különálló egységet kell tesztelni.

A Rails automatikusan minden vezérlő számára elkészíti a helper modult az az ***app/helpers*** könyvtárban.

## Linkek megadása

A linkek egyszerű elhelyezésére használható a ***link_to helper***, amely a megadott ***:controller, :action*** opciókból előállítja egy esemény URI-jét. Lehetséges hogy az linkeknek egyedi neveket adhatjuk, útvonal_név_path formában.

```
<%= link_to "Delete" , { :action => "delete" ,
                         :id =>@user}%>
```

## Űrlapok

Az Űrlapok készítése a leggyakoribb beviteli mód hogy a felhasználótól adatok kérjünk be, így a Rails biztosít ehhez is eszközt, az Action View FormHelper osztálya kezeli az űrlapokat. Ha egy olyan űrlapot szeretnénk ami csak egy Active Record modellen alapszik akkor ***form_for*** helpert célszerű használni.

```
<% form_for :user,:url =>{:action => :create} do |form| %>
  <p>Name:  <%= form.text_field :name, :size => 30 %></p>
  <p>Bio: <%= form.text_area :bio, :rows => 3 %></p>
  <p>Image URL:   <%= form.text_field :image_url %></p>
  <p><%= submit_tag %></p>
<% end %>
```

A ***form_for*** blokkján belül egyedi hatókört jön létre a megadott objektum körül, így az átadott blokkban úgy definiálhatjuk az elemeit, hogy csak a modell attribútumára kell hivatkoznunk, magára a modellobjektumra nem.

A ***fields_for*** és a ***form_for*** abban hasonlít hogy hasonlóan egy egyedi hatókört hoz létre a megadott modell objektum körül, de azzal ellentétben a form elemet nem hozza létre vele. A ***field_for*** arra való hogy a blokkjában több objektumot is lehessen hivatkozni mivel a ***form_for*** csak egyre ad lehetőséget.

## Partial fájlok

Webes alkalmazások készítésekor gyakran kell egy ugyanarról az objektumról több

oldalon adatokat megjeleníteni, ezek általában az átlagos keretrendszerekkel létrehozott alkalmazásoknál egy kód több helyen való elhelyezéséhez vezetne ami sértené a DRY alapelvet.

Ezt a kód duplikációt a Rails partial fájlok használatával küszöböli ki, ami rövid paraméterezhető sablon fájlok amit elhelyezhetünk más sablon fájlokban. Ezek általában egy objektumot írnak le amit többször használunk, a fájlok nevei aláhúzással kezdődnek, így a keretrendszer tudja melyik fájlt keresse.

A partial fájlok meghívását a sablonból a ***render*** metódussal lehet megtenni amit ki kell egészíteni a ***:partial*** opcióval, itt a megadhatjuk a partial fájl nevét és elérését, ilyen sablonok meghívásakor át kell adnunk valamilyen paramétert ezt a ***:locals*** változóval lehet elérni.

```
render(:partial => 'article' , :object => @an_article,
     :locals => { :authorized_by => session[:user_name]
     :from_ip => request.remote_ip })
```

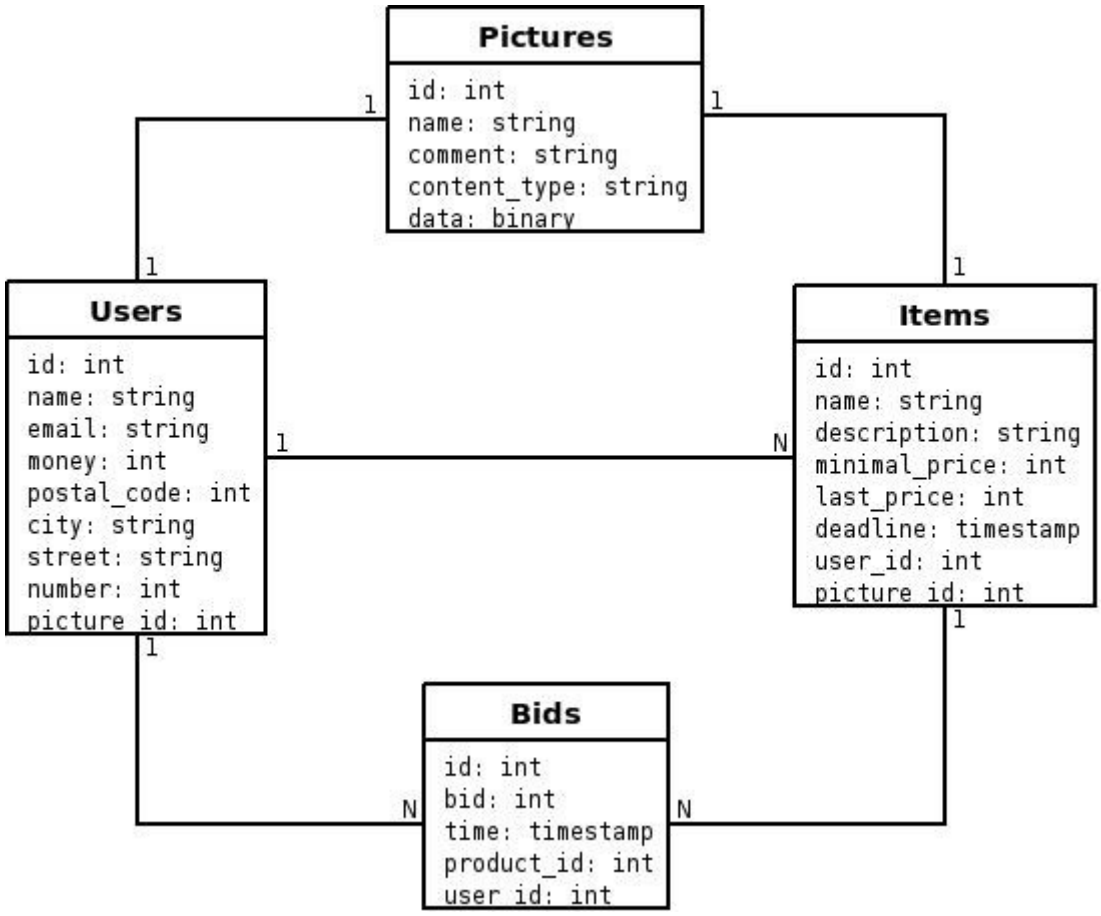
# 5. Esettanulmány, Black Market

## 5.1 Alkalmazás feladata és célja

Az alkalmazáson egy egyszerű aukciós oldal, ahova a felhasználók az eladandó tárgyaik adatait feltehetik és a többi felhasználó vagy látogató ezeket megtekinthetik. Ahhoz hogy egy adott tárgyra valaki licitálhasson ahhoz először regisztrálni kell.

## 5.2 Adatbázis kapcsolatok és felépítés

Az adatbázis négy fő táblázatból áll ami az a felhasználók adatait rögzíti, a tárgyak és a hozzátartozó paramétereket, áll egy táblából ami a liciteket tartalmazza és egy táblából ami a felhasználók és tárgyakhoz feltöltött képeket tartalmazza.




*ábra 3: A táblák közötti kapcsolatok*

A táblák közötti kapcsolatok külső kulcsokkal vannak fenntartva. Amit a Rails

ORM megközelítése egyszerűvé tesz, csak a modellek elején kell kell jelezni a kapcsolat típusát.

Az alkalmazásom alapjait a ***scaffold*** nevű Rails szkript segítségével hoztam létre, a scaffold segítségével létrehozhatjuk a modellt, a vezérlőt, a hozzátartozó nézetet és az útvonal beállításokat is. A ***scaffold*** nevű parancs a paraméterként megadott feltételek alapján először egy migrációt hoz létre amivel az adatbázist tudjuk létrehozni, az adatbázis neve ***scaffold*** parancs első paramétere, attribútumai pedig a további paraméterek. Ezek alapján létrehozza modellt is, ami reprezentálja az osztályt. Továbbá létrehozza a REST alapján a modell szerkesztéséhez, törléséhez, létrehozásához szükséges nézeteket a megfelelő vezérlővel amivel el tudjuk érni.

```
ruby script/generate scaffold cikk cim:string torzs:text
```

## 5.3 Felhasználói jogkörök és beléptetés

Az alkalmazást használók két csoportra bonthatók, az egyiket a felhasználók alkotják akik az oldalt regisztrált felhasználói, jogosultságuk van a tárgyak böngészésére, licitálására, saját profiljuk szerkesztésére. A másik csoport az adminisztrátorok akik törölhetnek, szerkesztethetnek, létrehozhatnak felhasználókat, tárgyakat, liciteket. Illetve még vannak a látogatók de ők csak a tárgyakat böngészhetik.

A beléptetésnél eldől a felhasználó jogköre, minden felhasználóhoz egy változót rendelünk az adatbázisban ami ezt a jogkört tartalmazza. Az alkalmazás működése közben a felhasználó azonosítóját és jogkörét elmentjük a ***session*** nevű globális változóba, így lehetséges a műveleteket korlátozni jogkör szerint.

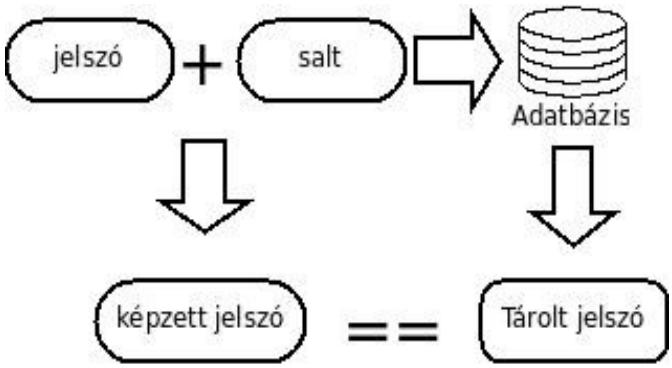
A Rails munkameneteihez tartozik a ***flash*** nevű hash tömb aktív használata. Bizonyos esetekben ugyanis szükségünk van arra, hogy üzeneteket tudjunk átadni két átirányított esemény között, például hogy hibaüzeneteket vagy információkat közöljünk a felhasználóval.

Azonban mivel minden átirányítás során a Rails új vezérlő objektumokat hoz létre és így az előzőleg létrehozott példányváltozók elvesznek, ezeket kénytelenek vagyunk a

munkamenetben elhelyezni, majd minden új híváskor törölni az eredeti üzenetet. A Rails ezt a ***flash*** nevű változó használatával könnyíti meg.

A ***flash*** változót ugyanúgy használhatjuk, mint a ***session*** változót, azzal a különbséggel, hogy a benne tárolt adat mindösszesen egy válaszig őrzi meg tartalmát, utána automatikusan törlődik. A ***flash*** rendelkezik továbbá két segéd metódussal: ***flash#new[]*** és ***flash#keep()***. Az előbbi a ***flash*** tartalmát csak az éppen futó eseményig tárolja el, nem adja át azt egy következőnek a munkamenetben. Míg utóbbi megőrzi a már létező flash tartalmat a következő átirányításig is.

Az adatbázisban nem tároljuk a jelszavakat eredeti formájukban, a jelszavakat kódolva tároljuk. Ezt a kódolt jelszót úgy hozzuk létre hogy a felhasználó jelszavához adjuk hozzá egy véletlenszerűen generált stringet, amit ***salt***-nak hívnak. Ezt egyirányú kódolásnak hívják. Ebből látható hogy az adatbázisban mind a titkosított jelszót, mind a véletlenszerű stringet is le kell tárolni, hiszen a beléptetésnél majd a felhasználó jelszavát újra kódoljuk a ***salt*** segítségével hogy összehasonlíthassuk a letárolt kódolt jelszóval.




*ábra 4: A jelszó összehasonlításának szemléltetése*

## 5.4 Aukció menete

Ahhoz hogy egy tárgyra licitálhasson a felhasználó ahhoz be kell lépni, a tárgyak rendelkeznek egy minimális licit és az aktuális licit attribútummal, ezek szerint a licitálást

szűrni kell, nem lehet érvényes egy licit ha kisebb értéket adtunk meg, valamint nem licitálhat a felhasználó nagyobb összeggel mint ami rendelkezésére áll. A határidő lejárta után értesítjük a felhasználót ha sikeres volt az aukciója.

## 5.5 Alkalmazás lokalizációs beállítása

A Rails alkalmazások lokalizációja sokáig nem volt beépítve, külön RubyGem kellett hozzá, a 2.2 verziószám óta be van építve a keretrendszerbe.

Ahhoz hogy ezt megtegyük meg be kell állítanunk néhány paramétert mert alapesetben a Rails az angol nyelvet részesíti előnyben. Meg kell adnunk a lokalizációs fájlok helyét és nevét. Alapesetben a ***config/locales*** könyvtárban keresi a lokalizációs fájlokat amit automatikusan be is tölt. Ezeket a ***I18n.load_path***, ***I18n.default_locale*** metódussal lehet felülírni, az elsővel a lokalizációs fájlok helyét a másodikkal az alapértelmezett nyelvet lehet beállítani. Alkalmazáson angol és magyar nyelven is elérhető.

A dátumok, időpontok formázásához a ***to_s*** metódussal megkapott stringet adjuk át paraméterként a ***strftime*** függvénynek amihez sok formázási string használható, egyszerűbb ha a formátumot lementjük a lokalizációs fájlban így csak egyszer kell átírnunk ha módosítani akarjuk.

# 6. Összefoglalás

A szakdolgozatom azt igyekezte megmutatni hogy a Ruby on Rails egy kiforrott keretrendszer ami viszonylag új, fiatal keretrendszerként folyamatosan fejlődik és felhasználói tábora is növekszik, a Ruby on Rails most az egyik leggyakrabban használt keretrendszerré vált a web 2.0 alkalmazások körében, legfőképp a Prototype és Script.aculo.us Javascript könyvtárak miatt.

Az alkalmazás készítése közben a Ruby kód használata miatt viszonylag hosszú után is érthető volt, dokumentációra szinte nem is volt szükségem. Az Active Record az adatbázis műveleteket nagy részben leegyszerűsítette, nagyban segítette a munkát legfőképp a migrációk, amivel könnyen lehetett változtatni az adatbázis sémát is.

Az eddig tapasztalataim alapján melyek során webes alkalmazást kellett készíteni eddig a Ruby on Rails volt az ami a leginkább magába integrálta az összes olyan funkciót amit egy mai igényeknek megfelelő oldal elkészítéshez szükséges. Mivel az eszközöket nem külön kellett beépíteni ezért sokkal könnyebben lehetett használni, kompatibilitási gondok nem igazán jellemzők mivel a részegységek az egész szerves részét képezik.

A Ruby on Raily használata során megismerkedtem az agilis metodikával, a dinamikus, szkript nyelvek előnyeivel, és sok újszerű szemlélettel melyet a Ruby on Rails remekül ötvöz és nem elhanyagolható módon működik a gyakorlatban.

# Irodalomjegyzék


[1] Sam Ruby , Dave Thomas, David Heinemeier Hansson: Agile Web Development with Rails, 3rd Edition, 2009
357-362. old., 397-407. old.

[2] Fábián Gergely: E-learning tananyag a Ruby on Rails keretrendszerhez, 2009
http://rails.ruby-oktatas.hu/chapters/tutorial/scaffolding

[3] A.P. Rajshekhar: Building Dynamic Web 2.0 Websites with Ruby on Rails 2008,
18-21. old., 52-58. old.

[4] Patrick Lenz : Simply Rails 2 , 2008,
276-280 old., 293-302. old

[5] Rails Framework Documentation
http://api.rubyonrails.org/, 2009

[6] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, 1994

[7] Dave Thomas: Programming Ruby, The Pragmatic Programmer's Guide
http://www.rubycentral.com/pickaxe/language.html#UJ

[8] Leonard Richardson, Sam Ruby: RESTful Web Services, 2007